

FILE COPY

ESD ACCESSION LIST

DRI Call No. 84863

Copy No. 1 of 2 copies  
RESEARCH ON AUTOMATIC PROGRAMMING

President and Fellows of Harvard College  
Cambridge, MA 02138

December 1975

Approved for Public Release;  
Distribution Unlimited.

Prepared for

DEPUTY FOR COMMAND AND MANAGEMENT SYSTEMS  
ELECTRONIC SYSTEMS DIVISION  
HANSCOM AIR FORCE BASE, MA 01731



ADA024335

### LEGAL NOTICE

When U. S. Government drawings, specifications or other data are used for any purpose other than a definitely related government procurement operation, the government thereby incurs no responsibility nor any obligation whatsoever; and the fact that the government may have formulated, furnished, or in any way supplied the said drawings, specifications, or other data is not to be regarded by implication or otherwise as in any manner licensing the holder or any other person or conveying any rights or permission to manufacture, use, or sell any patented invention that may in any way be related thereto.

### OTHER NOTICES


Do not return this copy. Retain or destroy.

This technical report has been reviewed and is approved for publication.

  
GEORGE E. REYNOLDS  
Engineering Planning Division (MCIO)

  
CHESTER G. CLARK, Lt. Col., USAF  
Chief, Engineering Planning Division (MCIO)

FOR THE COMMANDER

  
FRANK J. EMMA, Colonel, USAF  
Director, Information Systems  
Technology Applications Office  
Deputy for Command & Management Systems

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

DD FORM 1473 EDITION OF 1 NOV 65 IS OBSOLETE

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)



## SECTION I

### TECHNICAL SUMMARY

This report describes the results of the program of research carried out at Harvard University during the period October 1, 1973 through August 31, 1975 utilizing the PDP-10 and various resources on the ARPANET during that period.

The work was divided into three major areas: Automatic Programming Research and Tools for Software Production; Data Storage and Transfer; and, Operating System Research.

In the first area, our major results have been related to the enhancement of the ECL Programming System and the utilization of ECL as the basis for tools for software production. We have also experimented with a number of applications of ECL. Finally, a number of efforts have been concerned with various aspects of the question of program optimization.

During the period considerable use has been made of the ARPANET for data storage and transfer; a major programming system has been developed (PPL on the PDP-11/45) using a variety of tools and resources at several sites on the net.

Our efforts in the operating systems area have been concentrated primarily on devising system architectures for higher level languages.

Finally, the last few months have seen the beginning of design work on various components of a Program Manipulation System -- a system devised to integrate the various tools for program development, optimization, and maintenance.

- - - - -

This research was undertaken pursuant to contract F19628-74-C-0083 with Air Force Electronic Systems Division, Hanscom Air Force Base, Bedford, Massachusetts.



## SECTION II

### INTRODUCTION

This report describes the results of the program of research carried out at Harvard University during the period October 1, 1973 through August 31, 1975 utilizing the PDP-10 and various resources on the ARPANET during that period.

The major area of effort has been that of developing tools for software production and investigating various aspects of automatic programming, particularly those having to do with program transformation and optimization. During the period the ECL system has been considerably enhanced and a number of new tools developed based on ECL. The system has also been used for production problems by at least one government agency.

A second area of interest has been in the use of the ARPANET and various resources for data storage and transfer available at several locations on the net. Perhaps the major accomplishment here was in tying together the resources available at several sites to provide the tools which we used to implement a major programming system -- a full PPL system for the PDP-11/45 computer.

The third area, that of operating systems research, has been concerned primarily with the question of machine architectures for higher level languages.

The sections following are devoted to the various research areas in which we have been working in the course of the contract period. In each we have attempted to provide a brief review of the problems with which we have been concerned, cite the accomplishments during the contract period, and provide reference to the Bibliography where the reports, papers, and memoranda which contain the detailed results are listed. Copies of these documents are available upon request from the Librarian of the Center for Research in Computing Technology.

## SECTION III

### AUTOMATIC PROGRAMMING AND TOOLS FOR SOFTWARE PRODUCTION

#### 1. Construction of Tools

##### 1.1 Development of the ECL System

###### General Improvements

As part of the general effort to improve the usability of ECL for other tasks in this contract, a number of new features were added during this contract period. The error reporting and trapping facility was greatly expanded. Since 1974 the number of specific errors reported upon in detail has been increased from fifty to one hundred and twenty-four. Error messages of varying lengths keyed to each specific error were put into an error message file, expanding the text of the error messages from about 5000 characters to 23,000 characters. Appropriate user accessible parameters are also provided for each error. By revealing the precise nature of each error, these new diagnostics have greatly facilitated the rapid debugging of programs. In addition, they have been of great help to those learning the ECL language.

Direct access I/O routines were added to the system, thus allowing both random reading and writing of selected bytes of a file. These routines were of great utility in allowing the error message file to be indexed and the specific message for a given error to be rapidly accessed. Also, by remembering the byte position at which routines are defined in files to be compiled, it has now become possible to segment the compilation of large packages efficiently. The direct access I/O features were also of great use in the implementation of paged procedures.

In addition, a case statement, more powerful selection semantics, parallel declarations, and procedure mode validation were added, and these features documented in the new ECL manual [Manual]. The ECL system assembly language code was modularized, and monitor dependent code isolated in a single module in order to simplify the process of producing a version of ECL appropriate for use with the TENEX operating system used on many PDP-10s. A document describing the internal representation of ECL data types was produced to ease maintenance, and to aid in implementing ECL on other machines [Conrad 1].

###### (a) The ECL Compiler

The ECL compiler has been operational for over two years; it has been tuned and improved during the

contract period. Its new features are principally documented in [Manual] and [Opt-final], so they will only be summarized here.

A number of compiler options have been added to increase the efficiency of compiled code in situations where critical simplifying assumptions about the program or its data are not otherwise available. For example, expression data (list structures that can be both manipulated and evaluated) are quite common in ECL; however, they must normally remain uncompiled. A means has now been provided for specifying which expression data values will not be modified and may therefore be compiled. As another example, options now make possible the elimination of run-time checks during access of data components and pointer referents, so that thoroughly debugged production programs are not burdened with redundant type checking.

A variety of declarations about free variables may be used. For example a free variable can be "frozen" to given constant or shared values; if free variables are not "frozen" the user may still provide certain information about their modes and dynamic behavior. Other declarations allow the programmer to control the accessibility of local and global names in his production package, creating precisely the environment he wants its user to see.

Two methods now exist by which procedure calls may be eliminated during compilation for efficiency. The call may be replaced by an expression derived from the procedure body by substituting actual expressions for formal parameters. Or a user-defined routine may be invoked during compilation to supply a replacement derived from the given actual arguments. These substitution features are useful for efficient integration of user-defined mode behavior functions into compiled code.

An optional optimization phase has recently been added to the compiler. The optimizer is designed to remove the sorts of redundancy that most commonly arise in ECL programs, while staying within the strict resource limits that usually govern use of the compiler. Although ECL's semantic flexibility complicates the data flow analysis problems of redundancy removal, the optimizer incorporates a simple data invalidation detection scheme that enables significant improvements without elaborate analysis.

The user interface to the compiler (the collection of programs that help the programmer develop the global



declarations that bind his routines into a production package) has been expanded and smoothed. The use of ECL's program editor in this package building process has permitted the addition of features that make the use of the compiler much more straightforward. Several functions written using the editor's command repertoire have been provided for manipulating compiler declarations and checking them for consistency.

Using a new data module linking package, facilities have been added that permit compiling large programs in limited storage, either through separately compiled modules linked at loading time or by partitioning the program dynamically during compilation.

#### (b) User-Defined Data Types

User-defined data types were introduced into ECL in late 1972. They have been clarified and strengthened by the addition of several features during the present contract period. First, user-defined generation functions are now used consistently throughout the system in all instances of generation. Second, the user generation function is now used as a conversion function by taking conversion to be a case of generation by example. Third, through the addition of user-defined apparent dope vectors, it is now possible to model dimensioned data structures (e.g. matrices) using underlying representations which have different dimensionality (e.g. pointers). The double colon operator, which allows the specification of extended modes, was changed to take keywords followed by an argument to allow flexibility in defining the behavior of these modes.

#### (c) Free Storage Management

The compactifying garbage collector and the "fastmode" or "freebie count" storage allocation method, made possible by the compactifying garbage collector, became an integral part of the ECL system early in the contract period. It has become possible to handle much larger problems in a smaller period of time because the compactifying garbage collector enables "fragmentation", which results in a large number of relatively small blocks of storage but relatively few large blocks of storage, to be corrected by consolidating discontinuous free storage regions into one large contiguous free area. Fragmentation inhibits the allocation of large blocks of storage even though enough free space exists to accommodate them. With the compactifying garbage collector, it has been

possible to allocate pages of core storage without doing a garbage collection every time the storage free list becomes empty. This ability to allocate storage freely without damaging side effects has cut typical garbage collection time figures by a factor of from five to ten times [Conrad 2].

The compactifying garbage collector has also allowed the addition of expandable stacks to the ECL system. Since stack overflows can be trapped, and the stacks lengthened without losing the context of the program being run, the number of failures of programs to run because of stack overflow has been greatly reduced. The ability to adjust stack sizes dynamically is particularly important because it is difficult to determine in advance the stack requirements of a particular program used with varying sets of data.

A CEXPR (compiled procedure) paging mechanism was implemented in order to run larger programs in smaller core sizes. The basic method used was to prepare a "little brother" for each CEXPR to be paged; the "little brother" determines whether its "big brother" is already in core or whether it must be read in from a disk file. The "little brother" contains the file name and byte position of its "big brother" in the paging file. These permit the "big brother" to be brought rapidly into core, the recently implemented direct access I/O routines. Routines to delete CEXPRs not currently in use and to add the freed space to the system's free list without a garbage collection were also written. This paging mechanism has been successfully used to page the ECL editor, a very large program which requires only small subsets of its many routines to be in core at any given time.

The attempt to implement the Bobrow-Wegbreit control model revealed serious questions relating to its overall effect on the efficiency of the ECL system. Basically, the fact that new nomenclature (named variables) may be introduced in any ECL statement (as contrasted to LISP) leads to of very high overhead if the Bobrow-Wegbreit control model is directly implemented. The problem is particularly severe in compiled code. Since the introduction of a new named variable creates a new access environment that may need to be retained, the compiler's ability to optimize accesses would be greatly hampered. The matter is treated in more detail in the reference, "Bobrow-Wegbreit Control Model Reconsidered" [Conrad 3].



#### (d) SPECL Compiler

The SPECL (Systems Programming in ECL) project is intended to extend the use of ECL into areas normally reserved for so-called "implementation languages." SPECL is a dialect of ECL for which code can be compiled that runs without the support of ECL's runtime facilities. It offers the opportunity to "contract" ECL for special applications, since SPECL-produced code can be augmented by just the runtime support (such as storage management or I/O) that is actually needed.

SPECL also offers the user control over the actual code generated for built-in functions and permits him to control the selection of underlying data representations down to the level of machine words and bits, if he wishes. For example, suppose a programmer wants to implement doubly linked lists using a minimum of storage for the links. One trick is to give each element a single link field containing the bitwise Exclusive OR of the address of its predecessor with that of its successor. Given pointers to any two successive elements, it is then a simple matter to move forward or backward along the list.

SPECL is ideal for such an application because it permits the user to manage storage as he chooses and allows him to give machine code definitions for the necessary pointer operations. Access to the hardware level is isolated in code generation templates called Compiler Control Expressions (CCE). CCE's tell the compiler how to implement a given operator, depending on the states of its operands. The descriptions of operand states serve as goals for the code generator and they enable code selection for the various situations in which the corresponding user-specified operator will be used. Optimization of expressions containing user-specified operators is safe because the CCEs contain all the information about side effects and register use that the compiler needs to know.

The compiler consists of three phases. The first labels the program tree with temporary storage requirements (for subtrees free of common subexpressions) in a manner similar to the Sethi-Ullman method. Properties required for register allocation and assignment are attached to the program tree: lists of variables potentially invalidated by assignments, variables whose addresses may be tracked by register allocation, and a variety of minor properties necessary for code generation such as the number of locals declared in a block. An initial pruning of the possible code generation templates takes place as well.

The second phase operates on the tagged tree. It reorders computations to try to minimize the use of temporary storage. This step is necessary because the Sethi-Ullman algorithm does not allow for common subexpressions. The method is heuristic and incomplete because the minimization problem is polynomial complete.

Register allocation and assignment are also performed during the second phase. Like minimization, the assignment problem is polynomial complete, and it is handled heuristically.

The third phase simply translates the optimized program tree into object code.

Starting from a trial ordering of the expressions in each context (straight-line program section), the temporary-minimization procedure examines the variation in storage requirements over each context. The vicinities of peaks in the requirements are scanned for target positions that meet a simple numeric criterion based on the temporary usage and result size of the neighboring computations. Expressions are moved to these favorable positions in order to reduce overall storage use. Subtrees free of common sub-expressions move as units, subject to safety constraints. Most of the time used by this process is actually spent in recalculating the temporary requirements after an expression has been moved.

Register allocation begins with analysis of variables and common sub-expressions, first determining in what regions their values are relevant and then determining the minimum distance to next use of these items. This information suffices for optimal register allocation in straight-line code. Where branches are involved, however, a simple distance-to-next-use is not designated. Instead, the allocator must compare distances using a heuristic. The best one would include information about relative probabilities of taking one branch over another. Lacking such information the choice may be made on the basis of a simple average. Once the allocator determines the overall strategy of register allocation the assignment process must determine which register to use for each generation of a variable and must do it consistently at branches and join points of the program. The allocator must also allow for occasional specific register assignments. Heuristics are needed to avoid a "try all assignments" approach. At present, we have only simple algorithms for this problem. As usual, the issue is the trade-off between the cost of the algorithm and the



improvements in the code it makes.

In summary, then, SPECL extends to the hardware level the methodology that characterizes code improvement at higher levels. Users will be permitted to become involved in the optimization of their programs, and they will not need to forsake good structure to achieve highly efficient performance.

#### (e) ECL Model

The original definition of the semantics of the EL1 language [Wegbreit 2] was given as an EL1-coded version of the interpreter -- the so-called ECL model. By the fall of 1973, major revisions in the ECL system had rendered this definition out of date as well as incomplete. At that time, work was started to develop a new model.

There were a number of motivations for this work. First, as was the case in the thesis cited above, this method of modelling has proved to be valuable both in development and documentation. Partial models have been coded on several occasions:

1. An updated model of portions of the interpreter was prepared for instructional purposes during fall, 1973.
2. The revised mode compiler was modelled [Conrad 2] in early 1975.
3. Modelling was used extensively in preparing the revised edition of the ECL programmer's manual [Manual] in January 1975.
4. The parser and parser window were modelled during summer, 1975 [Sackut 1, 2, 3].

In the first two instances, the main purpose was pedagogical. In the second instance, however, the model preceded the implementation of the revision of declaration semantics. Models have been an important guide in the current development of the PDP-11/45 implementation of ECL.

In spring, 1974 we started a long-term effort to develop a model for the entire ECL system, not just for the interpreter. The major goal of the model is to serve as a guide for implementations of ECL on other hardware systems. Accordingly, much of the design effort in the model entails the attempt to isolate the machine-dependent aspects of the system (as has been

done in the machine language code for the PDP-10 implementation of the system) and, at the same time, to preserve maximum readability of the code. A major revision of the implementation strategy for the model took place in spring, 1975 to this end.

#### (f) Editor

The ECL list structure editor (EDIT) is an in-core editor which operates on the internal representation of uncompiled routines and unevaluated data rather than on character strings, as do conventional editors.

We have extended the editor to include a file editing facility (FIXIT). FIXIT allows a package consisting of one or more source files to be defined. It loads the package by parsing each file to construct a single block containing all of the commands in each source file in the package. At the same time, it relates each definition (that is, assignment command) to the corresponding source file. The source files are similarly related to the package so that several packages may be in core at the same time. This process is equivalent to loading, except that the properties allow the file to be reconstructed for output.

A routine, FIX, may be used at any time during debugging to apply EDIT to any routine loaded or to the form representing any file loaded by FIXIT. In the latter case, upon exit from the editor, FIX rebinds the definitions in the file in case new or revised definitions have been edited into the file form. Finally, the routine DUMPE, taking a package name as argument, will write out new versions of any files in the package that have been edited or any of whose definitions have been edited.

### 1.2 Programmable Theorem Prover

A number of issues relevant to developing a programmable theorem prover were investigated. These included techniques for specifying the behavioral properties of some data structure in an implementation independent manner, thus allowing verification of the correctness of concrete implementations of the data structure. These results are described in [Spitzen & Wegbreit].

### 1.3 Measurement Package

A number of additional mechanisms for measuring and probing ECL programs have been completed during the contract period. One is a procedure which takes the mode of a class

of data objects and returns the amount of space that an object from that class will require on a PDP-10. We have also extended the basic metering facility to permit a (non-redundant) measure to be made of an arbitrary collection of statements within a procedure. As part of this extension we can now produce a profile display of the metering results for greater readability. [Conrad 4]

#### 1.4 Closure Mechanism

The closure mechanism [Wegbreit 1] is a program transformation that replaces a procedure by an equivalent one in which the values of some formal parameters and free variables have been fixed and corresponding simplifications have been made.

It is a kind of optimization that becomes increasingly relevant as we learn more about "prefabricated construction" techniques for building large systems from standard components by tailoring them to special needs. It is well suited to program development by stepwise refinement, an implementation discipline that appears to enhance reliability and maintainability of programs.

Just before and during the early part of the contract period, we constructed a preliminary procedure closure mechanism. The intent was two-fold. It was to provide immediate benefit to ECL users as a practical code-improvement tool, and it was to give us insight into the nature of incremental optimization techniques. As a practical tool, that preliminary closure program was not a success. It proved awkward and very inefficient to use. It was not released to the general ECL user community, and in its stead several features were installed in the compiler. These included the ability to "freeze" free variables to constant or shared values or to fix their modes, facilities for in-line procedure expansion and invocation, and redundant expression optimization.

On the other hand, our experience with closure profoundly shaped our views of how the programming process can be automated. It has led us, in fact, to try to solve the problem by generalizing it. Some of the key difficulties raised by our study of the closure problem are:

- (a) The need for precise user guidance.

As a semi-automatic tool, closure requires that simplifying assumptions be supplied from which to derive specializations. The restriction to assertions about the free variables and formal parameters of an entire procedure was too severe. In practice, one needs to guide the simplifier to particular program



regions selected by pattern, by identifier scope, by the data types of constructs to be affected, and the like.

We have begun to satisfy this need for precise guidance in program transformation by the development of a Rewrite Mechanism [James] and by recent improvements in the ECL program editor [Manual]. The Rewrite Mechanism is a pattern-directed transformation facility that permits replacement rules to be associated with the lexical scopes of variables. These rules can embody simplifying assumptions and they themselves will be the object of simplifications (in a future system). The ECL editor is also pattern-directed, and it is now fully programmable, so that user-defined transformation routines can be applied to pattern-selected program regions.

(b) The need for program investigation aids.

Even though a program improvement tool may require user help to achieve significant results, it should not be assumed that the user is intimately familiar with the program being processed. In the important case that a library routine is being tailored to special requirements, quite the opposite may be true. Thus, facilities that help the user become familiar with a program package should be provided, so that he can be sure his simplifying assumptions about its behavior are valid.

As a step toward such a facility, we have implemented a program that studies free variable reference patterns in an ECL package [Jensen]. The information is displayed in a skeletal rendering of the program that also exhibits its recursive calling structure.

(c) The need for a data base supporting stepwise development.

Large-scale software seems to be most reliable and least expensive when it is produced in careful increments, each of which specializes some aspect of an abstract algorithm into more concrete terms. Optimization should be involved at each step, or vital information may be lost in the clutter of low level detail. If the program must be reanalyzed each time an optimizer is invoked, the cost is likely to be extremely high and disciplined implementation may disintegrate.



Thus the optimizer should be able to make incremental improvements without full reanalysis by maintaining a data base of information about the program acquired through user assertion, measurement, and analysis. Since implementation often proceeds by trial and error, the data base should include a history of the development and the user should be able to undo decisions, revise them, and then repeat unaffected portions of the implementation.

(d) The need for a common store of general-purpose optimization techniques.

The closure experiment was an attempt to achieve a limited class of code improvements with limited resources. It demonstrated the difficulty of trying to factor global program analysis into a set of independent tools without gross duplication of effort. A strong conclusion is that automatic programming tools should be integrated around a single, general-purpose semantic analyzer, so that all can share a common representation, a common data base, and a store of analytical and manipulative expertise.

To answer these needs, a new program manipulation system is under development. Its basis will be a full Symbolic Evaluator for ECL programs. This analyzer will maintain a Program Data Base to support incremental optimization and to preserve the refinement history of the user's program as its implementation progresses.

These facilities will not be cheap. Deep analysis of practical programs is costly to obtain and to maintain. On the other hand, a deep analysis can be viewed as an investment whose dividends will be the relative simplicity of adding such special-purpose tools as closure, better code generators to the kernel system for ECL and SPECL, and a data representation optimizer.

## 2. Applications

### 2.1 Parallel Processing

The SYNVER system (see [Griffiths 2]) has been developed during the contract period. The SYNVER system permits a high level specification of problems in concurrent control in the SAL specification language, which has been designed and implemented. The SYNVER synthesizer then generates ECL runnable code which enables concurrent processes to communicate with each other in the manner specified. A technique has now been developed whereby the correctness of the specification can be verified.

Code generated to effect the synchronization makes use of the ECL control extension facility (Prenner's CI, see [Prenner]). The adaptation to a code generator for Hoare's monitors (CACM, Oct. 1974) is written but not implemented. In addition, a code generator for Dijkstra's P and V semaphore operations [Dijkstra] is being developed. Initial results for this code generator are very encouraging; in many cases generated code is equivalent to that which would be hand-written.

The results of this work are described in [Griffiths 2].

## 2.2 Data Type Optimization

The extension to ECL to accommodate sets, tuples, stacks, and queues has been done. The extension includes new notational facilities (in the case of sets) for more natural specification of operations. We have used the mode behavior facility to specify the intended behavior of objects from the classes of data objects. In the case of sets we have implemented several different underlying machine representations for objects. In each case we have defined the necessary primitive operations. We have not yet solved the problem, however, of (mechanically) choosing efficient representations for objects, taking into consideration the access patterns and other primitive operations on the objects (e.g. insertions and deletions). This is a more difficult problem than anticipated, but it still promises high pay-off.

## 2.3 Compound Objects and Operations

We have developed the foundations for exploring the customizing of general purpose algorithms for the domain of matrices. We have implemented a package of basic general purpose operations and defined a collection of representations for matrices, with particular concern for sparse matrices. The package will use information about algebraic properties (in particular algebraic identities) of the matrices and do some optimization of the operations taking into account these properties. It will also define for the user some non-basic operations, such as transitive closure, permitted by the properties. The package includes a number of notational extensions to facilitate the user's writing of iterations (e.g.  $M[I, *]$  to specify the selection of the  $I$ th row) and partitions of a matrix. Several underlying representations have been implemented for sparse matrices, and we are now at the stage of considering the interaction among representations.

## 2.4 Resource Management

A linkage loading and dumping facility (LINK) which is more general than those in previous systems has been added to the ECL system. While the earlier version of the system allowed a package to be compiled in several modules, which were then loaded together for execution, linkage between different modules was effected by invocation of the interpreter. In order to eliminate this interpretation at run time, a reference to a routine external to the package being compiled may be replaced at load time by a pointer to a routine external to that particular compilation. This affords exactly as tight binding as would have been the case if the two routines had been compiled together.

External references are defined to the compiler by a compiler driver list, which is a list of pairs containing the identifiers and modes of external routines and/or data. Following compilation, the linkage dumper traces all compiled routines in order to locate all external references in the compiled code. The results of compilation are then dumped, together with tables which allow the linkage loader to link separately-compiled modules to each other and, where required, to routines and data which are already loaded.

The link dumper may, in fact, be used to separate a mass of loaded routines and data into separate files. These files will include the information required for linkage loading, independent of use of the compiler. Due to the manner in which routines and data are represented in core, such separation can be effected for any data objects which are bound to identifiers at top level or on the name stack. Thus, for instance, if a data table consisted of, among other things, a sequence of routines, different routines might be dumped into different files with the resulting files being reloadable. A forthcoming version of the package will extend this behavior to arbitrary data objects including those which are not necessarily related to a compilation.

## 3. Related Research Efforts

### 3.1 Property Extraction

Property extraction is a generic name for a set of techniques for efficiently extracting from some set of programs a collection of properties which enable program transformation and optimization. During the contract period a prototype system to explore the technique was developed and reported in [Scherlis]. Work is continuing.



## SECTION IV

### DATA STORAGE AND TRANSFER

#### 4.1 Use of the DATACOMPUTER

Experience with the use of the DFTP (Datacomputer File Transfer Program) developed by Computer Corporation of America (CCA) has shown that a more automatic method of file archiving and retrieval, while feasible, is probably not worth the development effort required. We have also investigated the feasibility of referencing files stored on the Datacomputer from ECL programs. While this facility is easily implementable in ECL, given recent improvements in the input-output and error-handling facilities, the result would be to overburden the currently limited number of network ports on CCA's TENEX computer at very little gain to our own operation. Therefore, no actual development work has been undertaken in this area.

#### 4.2 Datacomputer and Datalanguage Development

Except for several consultations with CCA, no work has been done in this area. Our current attitude with regard to conversion of data stored on the Datacomputer is that, in the absence of economic justification in the case of any particular application, the actual computations required should take place at other sites. Such conversions may be done with significantly less overhead by a facility which knows both the source and target data types than is possible at a centralized facility which must deal with a far larger set of data types and formats.

#### 4.3 Graphics Studies

The development of a sophisticated graphics system was undertaken. The objective of this effort was to experiment with the viability of graphics processors with graphics instructions sets capable of directly interpreting a high level data structure to produce graphical output. This approach is to be contrasted with the classical real-time compilation of a data structure into some low-level object format suitable for execution by a graphics controller.

A system that would support research in micro-programmed graphics instruction sets was designed and partially implemented. It consisted of a graphics processor (PDP-11/40) linked to a local processor (PDP-11/10) and a PDP-10 host machine. The PDP-11/40 was extensively modified to be a dynamically micro-programmable graphics processor. We added a writeable control store, hardware micro-program development aids, and (partially implemented) registers accessible at the micro-program level which drive display



hardware. A special interface allows the two PDP-11s to share the same address space. A high-speed parallel channel was built to permit fast access to the PDP-10.

Prints covering the installed hardware are available, as well as listings of associated software. There were no patentable inventions which resulted.

## SECTION V

### SYSTEM ARCHITECTURE FOR HIGH-LEVEL LANGUAGE

The problem of implementing complex computations as embodied by, say, high-level languages has customarily been attacked in relation to classical machine architectures. These architectures have been and persist in being rather arbitrary in that they do not, in any formalized way, reflect the nature of the computations that are anticipated.

The problem of constructing machine architectures that are appropriate to specific computational problems, (in particular EL1 programs), is currently being investigated. The objective of this effort is to develop formal methods by means of which a system designer may take a representation of a desired computation and transform it into an implementation that capitalizes on the specific nature of the computation.

The approach being pursued represents a computation within a formalism that introduces no overhead and exhibits all potential concurrency. This representation, resembling an infinite Petri net, is then subjected to a series of transformations that reduce it to a finite form, and in particular, a form with properties making it readily implementable by some set of hardware primitives. Each transformation is chosen by the system designer and has a quantitative effect on performance that may be readily perceived.

This research is aimed at developing a system that achieves representational and execution efficiency by using EL1 as the lowest level of machine language. We seek to determine the facilities and characteristics such a system should possess as a whole.

This work is reported on in [Marcuvitz].

## BIBLIOGRAPHY

- [Broszgol] Broszgol, Benjamin M., Deterministic translation grammars, Ph.D. Thesis, Harvard University, November 1973; Center for Research in Computing Technology, TR-3-74.
- [Byrn] Byrn, William H., Sequential processes, deadlocks, and semaphore primitives, Ph.D. Thesis, Harvard University, November 1974; Center for Research in Computing Technology, TR-7-75.
- [Cheatham] Cheatham, Thomas E., Jr., The unexpected impact of computers on science and mathematics, Harvard University, Center for Research in Computing Technology, TR-27-74, December 1974.
- [Cheatham & Townley 1] Cheatham, Thomas E., Jr., and Judy A. Townley, A proposed system for structured programming, Harvard University, Center for Research in Computing Technology, TR-11-74, May 1974.
- [Cheatham & Townley 2] Cheatham, Thomas E., Jr., and Judy A. Townley, A look at programming and programming languages, Harvard University, Center for Research in Computing Technology, TR-18-75, June 1975.
- [Cohen & Taft] Cohen, Dan, and Edward Taft, Harvard network graphics systems, Working paper, Harvard University, Center for Research in Computing Technology, TR-4-74, November 1973.
- [Conrad 1] Conrad, William R., A compactifying garbage collector for ECL's non-homogeneous heap, Harvard University, Center for Research in Computing Technology, TR-2-74, January 1974.
- [Conrad 2] Conrad, William R., Internal representations of ECL data objects, Harvard University, Center for Research in Computing Technology, TR-5-75, February 1975.
- [Conrad 3] Conrad, William R., Bobrow-Wegbreit control model reconsidered, Center for Research in Computing Technology, Harvard University, March, 1975, CR-3-75.
- [Conrad 4] Conrad, William R., PROBE Timing Algorithms, Harvard University, Center for Research in Computing Technology, December 1975, CR-12-75.
- [Dijkstra] Dijkstra, W.W., Cooperating Sequential Processes, in Programming Languages, (F. Geneey, ed.), Academic Press, New York, 1968, pp. 43-112.



- [German] German, Steven M., A program verifier that generates inductive assertions, Harvard University, Center for Research in Computing Technology, TR-19-74, May 1974.
- [Griffiths 1] Griffiths, Patricia, SAL: a very high specification language, Harvard University, Center for Research in Computing Technology, TR-25-74, November 1974.
- [Griffiths 2] Griffiths, Patricia, SYNVER: a system for the automatic synthesis and verification of synchronization processes, Harvard University, Center for Research in Computing Technology, TR-22-74, September 1974.
- [James] James, Bernard, A rewrite mechanism for stepwise refinement, Center for Research in Computing Technology, Harvard University, April 1975, CR-4-75.
- [Jensen] Jensen, Philip, Finding free variables in ECL programs, Center for Research in Computing Technology, Harvard University, May 1975, CR-5-75.
- [Manual] ECL Programmer's Manual, revised edition. Harvard University, Center for Research in Computing Technology, TR-23-74, September 1974.
- [Marcuvitz] Marcuvitz, Andrew, The impact of a high-level language on systems architecture, Center for Research in Computing Technology, February 1974, CR-2-74.
- [Mealy] Mealy, George H., Data structures: theory and representation, Harvard University, Center for Research in Computing Technology, TR-10-75, May 1975.
- [Opt-final] Final report on research in optimization techniques. Harvard University, Center for Research in Computing Technology, June 1975; ESD-TR-75-81.
- [Prenner] Prenner, Charles, J., Multi-Path Control Structures for Programming Languages. Ph.D. Thesis, Harvard University, June 1972; ESD-TR-70-30; Harvard University, Center for Research in Computing Technology, TR-10-72.
- [Scherlis] Scherlis, W.L., On the weak interpretation method for extracting program properties. Harvard University, Center for Research in Computing Technology, May 1974, CR-5-74.

- [Shostak] Shostak, Robert E., Refutation graphs and resolution theorem proving, Working paper, Harvard University, Center for Research in Computing Technology, TR-1-74, January 1974.
- [Sockut 1] Sockut, Gary, The new parse window, Harvard University, Center for Research in Computing Technology, October 1975, CR-9-75.
- [Sockut 2] Sockut, Gary, Using and compiling the parser table generator, Harvard University, Center for Research in Computing Technology, October 1975, CR-10-75.
- [Sockut 3] Sockut, Gary, Using the ECL model's lexical analyzer and parser, Harvard University, Center for Research in Computing Technology, October 1975, CR-11-75.
- [Spitzen] Spitzen, Jay M., Approaches to automatic programming, Ph.D. Thesis, Harvard University, June, 1974; Center for Research in Computing Technology, TR-17-74.
- [Spitzen & Wegbreit] Spitzen, Jay M., and Ben Wegbreit, The verification and synthesis of data structures. Acta Informatica, 4, 127-144 (1975).
- [Wegbreit 1] Wegbreit, B. Procedure closure in EL1, Computer Journal, 17, 38-43, (Feb. 1974).
- [Wegbreit 2] Wegbreit, B. Studies in extensible languages. Ph.D. Thesis, Harvard University, June 1970; ESD-TR-70-297; Harvard University, Center for Research in Computing Technology, TR-3-72.